Readers-Writers Problem

- Recall a mutex allows in only one thread
- But a data race occurs only if
  - multiple threads access the same data, and
  - at least one of the accesses is a write
- How to allow multiple readers or one single writer?
  - Need lock that can be shared amongst concurrent readers
- Can implement using other primitives (next slide)
  - Keep integer \( i \) – # or readers or -1 if held by writer
  - Protect \( i \) with mutex
  - Sleep on condition variable when can’t get lock

Implementing shared locks

```c
struct sharedlk {
  int i; /* # shared lockers, or -1 if exclusively locked */
  mutex_t m;
  cond_t c;
};

void AcquireExclusive (sharedlk *sl) {
  lock (sl->m);
  while (sl->i) { wait (sl->m, sl->c); }
  sl->i = -1;
  unlock (sl->m);
}

void AcquireShared (sharedlk *sl) {
  lock (sl->m);
  while (sl->i < 0) { wait (sl->m, sl->c); }
  sl->i++;
  unlock (sl->m);
}

void ReleaseShared (sharedlk *sl) {
  lock (sl->m);
  if (!--sl->i) signal (sl->c);
  unlock (sl->m);
}

void ReleaseExclusive (sharedlk *sl) {
  lock (sl->m);
  sl->i = 0;
  broadcast (sl->c);
  unlock (sl->m);
}
```

• Note: Must deal with starvation

Review: Test-and-set spinlock

```c
struct var {
  int lock;
  int val;
};

void atomic_inc (var *v) {
  while (test_and_set (&v->lock))
    ;
  v->val++;
  v->lock = 0;
}

void atomic_dec (var *v) {
  while (test_and_set (&v->lock))
    ;
  v->val--;
  v->lock = 0;
}
```

• Is this code correct without sequential consistency?

Memory reordering danger

- Suppose no sequential consistency & don’t compensate
- Hardware could violate program order

<table>
<thead>
<tr>
<th>Program order on CPU #1</th>
<th>View on CPU #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write: v-&gt;lock = 1;</td>
<td>v-&gt;lock = 1;</td>
</tr>
<tr>
<td>read: register = v-&gt;val;</td>
<td>v-&gt;lock = 0;</td>
</tr>
<tr>
<td>write: v-&gt;val = register + 1;</td>
<td>v-&gt;val = register + 1;</td>
</tr>
<tr>
<td>write: v-&gt;lock = 0;</td>
<td>/* danger */</td>
</tr>
</tbody>
</table>

• If atomic_inc called at /* danger */, bad val ensues!
Ordering requirements

void atomic_inc (var *v) {
    while (test_and_set (&v->lock))
    ;
    v->val++;
    /* danger */
    v->lock = 0;
}

• Must ensure all CPUs see the following:
  1. v->lock was set before v->val was read and written
  2. v->lock was cleared after v->val was written
• How does #1 get assured on x86?
  - Recall test_and_set uses xchgl %eax,(%edx)
  - xchgl instruction always “locked,” ensuring barrier
• How to ensure #2 on x86?
  - Might need fence instruction after, e.g., non-temporal stores

Correct spinlock on alpha

• ld1_l – load locked
  c
stl_c – store conditional (sets reg to 0 if not atomic w. ld1_l)

_test_and_set:
    ldq_l v0, 0(a0) # v0 = *lockp (LOCKED)
    bne v0, 1f # if (v0) return
    addq zero, 1, v0 # v0 = 1
    stq_c v0, 0(a0) # *lockp = v0 (CONDITIONAL)
    beq v0, _test_and_set # if (failed) try again
    mb
    addq zero, zero, v0 # return 0
1: ret zero, (ra), 1

• Note: Alpha memory consistency much weaker than x86
• Must insert memory barrier instruction, mb (like mfence)
  - All processors will see that everything before mb in program order
  happened before everything after mb in program order

Memory barriers/fences

• Must use memory barriers (a.k.a. fences) to preserve program
  order of memory accesses with respect to locks
• Many examples in this lecture assume S.C.
  - Useful on non-S.C. hardware, but must add barriers
• Dealing with memory consistency important
  - See [Howells] for how Linux deals with memory consistency
  - C++ now exposes support for different memory orderings
• Fortunately, consistency need not overly complicate code
  - If you do locking right, only need to add a few barriers
  - Code will be easily portable to new CPUs

Outline

1 Synchronization and memory consistency review
2 Cache coherence – the hardware view
3 Avoiding locks
4 Improving spinlock performance
5 Kernel interface for sleeping locks
6 Deadlock
Cache coherence

- Performance requires caches
- Sequential consistency requires cache coherence
- Barrier & atomic ops require cache coherence
- Bus-based approaches
  - “Snoopy” protocols, each CPU listens to memory bus
  - Use write through and invalidate when you see a write
  - Or have ownership scheme, e.g., MESI (MESIF, MOESI, ...) bits
  - Bus-based schemes limit scalability
- Cache-Only Memory Architecture (COMA)
  - Each CPU has local RAM, treated as cache
  - Cache lines migrate around based on access
  - Data lives only in cache

NUMA and spinlocks

- Test-and-set spinlock has several advantages
  - Simple to implement and understand
  - One memory location for arbitrarily many CPUs
- But also has disadvantages
  - Lots of traffic over memory bus (especially when > 1 spinner)
  - Not necessarily fair (same CPU acquires lock many times)
  - Even less fair on a NUMA machine
  - Allegedly Google had fairness problems even on Opterons
- Idea 1: Avoid spinlocks altogether
- Idea 2: Reduce bus traffic with better spinlocks
  - Design lock that spins only on local memory
  - Also gives better fairness

Recall producer/consumer (lecture 3)

```c
/* PRODUCER */
for (;;) {
    item *nextProduced = produce_item();
    mutex_lock (&mutex);
    while (count == BUF_SIZE)
        cond_wait (&nonfull, &mutex);
    buffer [in] = nextProduced;
    in = (in + 1) % BUF_SIZE;
    count++;
    cond_signal (&nonfull);
    mutex_unlock (&mutex);
}
```

```c
/* CONSUMER */
for (;;) {
    mutex_lock (&mutex);
    while (count == 0)
        cond_wait (&nonempty, &mutex);
    nextConsumed = buffer[out];
    out = (out + 1) % BUF_SIZE;
    count--;
    cond_signal (&nonfull);
    mutex_unlock (&mutex);
    consume_item (nextConsumed);
}
```

cc-NUMA

- Previous slide had dance hall architectures
- Any CPU can “dance with” any memory equally
- An alternative: Non-Uniform Memory Access
  - Each CPU has fast access to some “close” memory
  - Slower to access memory that is farther away
  - Use a directory to keep track of who is caching what
- Originally for machines with many CPUs
  - But AMD Opterons integrated mem. controller, essentially NUMA
  - Now intel CPUs are like this, too
- cc-NUMA = cache-coherent NUMA
  - Can also have non-cache-coherent NUMA, though uncommon
  - BBN Butterfly 1 has no cache at all
  - Cray T3D has local/global memory

Outline

1. Synchronization and memory consistency review
2. Cache coherence – the hardware view
3. Avoiding locks
4. Improving spinlock performance
5. Kernel interface for sleeping locks
6. Deadlock

Eliminating locks

- One use of locks is to coordinate multiple updates of single piece of state
- How to remove locks here?
  - Factor state so that each variable only has a single writer
- Producer/consumer example revisited
  - Assume you have sequential consistency
  - Assume one producer, one consumer
  - Why do we need count variable, written by both?
    - To detect buffer full/empty
  - Have producer write in, consumer write out
  - Use in/out to detect buffer state
  - But note next example busy-waits, which is less good
Lock-free producer/consumer

```c
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        while (((in + 1) % BUF_SIZE) == out)
            thread_yield ();
        buffer [in] = nextProduced;
        in = (in + 1) % BUF_SIZE;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (in == out)
            thread_yield ();
        nextConsumed = buffer[out];
        out = (out + 1) % BUF_SIZE;
        consume_item (nextConsumed);
    }
}
```

Non-blocking synchronization

- Design algorithm to avoid critical sections
  - Any threads can make progress if other threads are preempted
  - Which wouldn’t be the case if preempted thread held a lock
- Requires atomic instructions available on many CPUs
  - E.g., atomic compare and swap: CAS (mem, old, new)
    - If *mem == old, then set *mem = new and return true, else false
- Can implement many common data structures
  - Stacks, queues, even hash tables
- Can implement any algorithm on right hardware
  - Need operation such as atomic compare and swap
    (has property called consensus number = ∞ [Herlihy])
  - Entire kernels have been written w/o locks [Greenwald]
  - C++ now facilitates non-blocking algorithms w. atomic library

Example: stack

```c
struct item {
    /* data */
    struct item *next;
};
typedef struct item *stack_t;

void atomic_push (stack_t *stack, item *i) {
    do {
        i->next = *stack;
    } while (!CAS (stack, i->next, i));
}

item *atomic_pop (stack_t stack) {
    item *i;
    do {
        i = *stack;
    } while (!CAS (stack, i, i->next));
    return i;
}
```

Wait-free stack issues

- “ABA” race in pop if other thread pops, re-pushes i
  - Can be solved by counters or hazard pointers to delay re-use

Benign races

- Can also eliminate locks by having race conditions
- Sometimes “cheating” buys efficiency...
- Care more about speed than accuracy
  ```c
  hits++; /* each time someone accesses web site */
  ```
- Know you can get away with race
  ```c
  if (!initialized) {
      lock (m);
      if (!initialized) {
          initialize ();
          /* might need memory barrier here */
          initialized = 1;
      }
  }
  unlock (m);
  ```

Read-copy update [McKenney]

- Some data is read way more often than written
- Routing tables
  - Consulted for each packet that is forwarded
- Data maps in system with 100+ disks
  - Updated when disk fails, maybe every 10^10 operations
- Optimize for the common case of reading w/o lock
  - E.g., global variable: routing_table *rt;
  - Call lookup (rt, route); with no locking
- Update by making copy, swapping pointer
  - E.g., routing_table *nrt = copy.routing_table (rt);
  - Update nrt
  - Set global rt = nrt when done updating
  - All lookup calls see consistent old or new table
Garbage collection

- When can you free memory of old routing table?
  - When you are guaranteed no one is using it—how to determine

- Definitions:
  - temporary variable – short-used (e.g., local) variable
  - permanent variable – long lived data (e.g., global rt pointer)
  - quiescent state – when all a thread’s temporary variables dead
  - quiescent period – time during which every thread has been in quiescent state at least once

- Free old copy of updated data after quiescent period
  - How to determine when quiescent period has gone by?
    - E.g., keep count of syscalls/context switches on each CPU
    - Can’t hold a pointer across context switch or user mode
      (Preemptable kernel complicates things slightly)

MCS lock

- Idea 2: Build a better spinlock
- Lock designed by Mellor-Crummey and Scott
  - Goal: reduce bus traffic on cc machines, improve fairness
- Each CPU has a qnode structure in local memory
  
  ```
  typedef struct qnode {
    struct qnode *next;
    bool locked;
  } qnode;
  ```
  - Local can mean local memory in NUMA machine
  - Or just its own cache line that gets cached in exclusive mode
- A lock is just a pointer to a qnode
  
  ```
  typedef qnode *lock;
  ```
  - Lock is list of CPUs holding or waiting for lock
  - While waiting, spin on your local locked flag

MCS Acquire

```c
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (predecessor, *L); /* atomic swap */
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```
**MCS Acquire**

```c
acquire (lock *L, qnode *I) {
    I->next = NULL;
    qnode *predecessor = I;
    XCHG (predecessor, *L); /* atomic swap */
    if (predecessor != NULL) {
        I->locked = true;
        predecessor->next = I;
        while (I->locked)
    }
}
```

- If unlocked, *L is NULL
- If locked, no waiters, *L is owner's qnode
- If waiters, *L is tail of waiter list:

```c
predecessor

owner -> waiter -> waiter

*I

NULL
```

**MCS Release with CAS**

```c
release (lock *L, qnode *I) {
    if (!I->next)
        if (CAS (*L, I, NULL))
            return;
    while (!I->next)
        ;
    I->next->locked = false;
}
```

- If I->next NULL and *L == I
  - No one else is waiting for lock, OK to set *L = NULL
  ```c
  *L
  \[next\]
  \[NULL\]
  ```
- If I->next non-NULL
  - I->next oldest waiter, wake up w. I->next->locked = false
  ```c
  *L
  \[next\]
  \[NULL\]
  ```

**MCS Release w/o CAS**

- What to do if no atomic compare & swap?
- Be optimistic—read *L w. two XCHGs:
  1. Atomically swap NULL into *L
     - If old value of *L was I, no waiters and we are done
  2. Atomically swap old *L value back into *L
     - If *L unchanged, same effect as CAS
- Otherwise, we have to clean up the mess
  - Some "userper" attempted to acquire lock between 1 and 2
  - Because *L was NULL, the userper succeeded
    (May be followed by zero or more waiters)
  - Stick old list of waiters on to end of new last waiter

```c
release (lock *L, qnode *I) {
    if (I->next)
        I->next->locked = false;
    else {
        qnode *old_tail = NULL;
        XCHG (*L, old_tail);
        if (old_tail == I)
            return;
        qnode *userper = old_tail;
        XCHG (*L, userper);
        while (I->next == NULL)
            ;
        if (userper != NULL) {
            /* Someone changed *L between 2 XCHGs */
            userper->next = I->next;
        }
        I->next->locked = false;
    }
}
```

- If I->next is non-NULL
  - I->next oldest waiter, wake up w. I->next->locked = false
  ```c
  *L
  \[next\]
  \[NULL\]
  ```

**MCS Release w/o C&S code**

```c
release (lock *L, qnode *I) {
    if (!I->next)
        if (CAS (*L, I, NULL))
            return;
    else {
        qnode *old_tail = NULL;
        XCHG (*L, old_tail);
        if (old_tail == I)
            return;
        qnode *userper = old_tail;
        XCHG (*L, userper);
        while (I->next == NULL)
            ;
        if (userper != NULL) {
            /* Someone changed *L between 2 XCHGs */
            userper->next = I->next;
        }
        I->next->locked = false;
    }
}
```
Kernel support for synchronization

- Locks must interact with scheduler
  - For processes or kernel threads, must go into kernel (expensive)
  - Idea: only go into kernel if you can’t get lock

```
struct lock {
  int busy;
  thread *waiters;
};
void acquire (lock *lk) {
  while (test_and_set (&lk->busy)) { /* 1 */
    atomic_push (&lk->waiters, self); /* 2 */
    sleep ();
  }
}
void release (lock *lk) {
  lk->busy = 0;
  wakeup (atomic_pop (&lk->waiters));
}
```

Race condition

- Unfortunately, previous slide not safe
  - What happens if release called between lines 1 and 2?
  - wakeup called on NULL, so acquire blocks
- `futex` abstraction solves the problem [Franke]
  - Ask kernel to sleep only if memory location hasn’t changed
  ```
  void futex (int *uaddr, FUTEX_WAIT, int val...);
  - Go to sleep only if *uaddr == val
  - Extra arguments allow timeouts, etc.
  void futex (int *uaddr, FUTEX_WAKE, int val...);
  - Wake up at most val threads sleeping on uaddr
  - uaddr is translated down to offset in VM object
    - So works on memory mapped file at different virtual addresses in different processes
  ```

The deadlock problem

```c
mutex_t m1, m2;
void p1 (void *ignored) {
  lock (m1);
  lock (m2);
  /* critical section */
  unlock (m2);
  unlock (m1);
}
void p2 (void *ignored) {
  lock (m2);
  lock (m1);
  /* critical section */
  unlock (m1);
  unlock (m2);
}
```

- This program can cease to make progress – how?
- Can you have deadlock w/o mutexes?

More deadlocks

- Same problem with condition variables
  - Suppose resource 1 managed by c1, resource 2 by c2
  - A has 1, waits on c2, B has 2, waits on c1
- Or have combined mutex/condition variable deadlock:
  ```
  lock (a); lock (b); while (!ready) wait (b, c);
  unlock (b); unlock (a);
  lock (a); lock (b); ready = true; signal (c);
  unlock (b); unlock (a);
  ```
- One lesson: Dangerous to hold locks when crossing abstraction barriers!
  - I.e., lock (a) then call function that uses condition variable
Deadlocks w/o computers

- Real issue is \textit{resources} \& how required
- E.g., bridge only allows traffic in one direction
  - Each section of a bridge can be viewed as a resource.
  - If a deadlock occurs, it can be resolved if one car backs up
    (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

**Deadlock conditions**

1. Limited access (mutual exclusion):
   - Resource can only be shared with finite users
2. No preemption:
   - Once resource granted, cannot be taken away
3. Multiple independent requests (hold and wait):
   - Don’t ask all at once (wait for next resource while holding current one)
4. Circularity in graph of requests
   - All of 1–4 necessary for deadlock to occur
   - Two approaches to dealing with deadlock:
     - Pro-active: prevention
     - Reactive: detection + corrective action

**Prevent by eliminating one condition**

1. Limited access (mutual exclusion):
   - Buy more resources, split into pieces, or virtualize to make
     “infinite” copies
   - Threads: threads have copy of registers = no lock
2. No preemption:
   - Physical memory: virtualized with VM, can take physical page
     away and give to another process!
3. Multiple independent requests (hold and wait):
   - Wait on all resources at once (must know in advance)
4. Circularity in graph of requests
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)

**Resource-allocation graph**

- View system as graph
  - Processes and Resources are nodes
  - Resource Requests and Assignments are edges
- Process:
- Resource w. 4 instances:
- \(P_i\) requesting \(R_j\):
- \(P_i\) holding instance of \(R_j\):

**Example resource allocation graph**

**Graph with deadlock**
**Is this deadlock?**

- If graph has no cycles \(\implies\) no deadlock
- If graph contains a cycle
  - Definitely deadlock if only one instance per resource
  - Otherwise, maybe deadlock, maybe not
- **Prevent deadlock w. partial order on resources**
  - E.g., always acquire mutex \(m_1\) before \(m_2\)
  - Usually design locking discipline for application this way

---

**Cycles and deadlock**

- **Prevention**
  - Determine safe states based on possible resource allocation
  - Conservatively prohibits non-deadlocked states

- **Claim edges**
  - Dotted line is *claim edge*
    - Signifies process may request resource

- **Example: unsafe state**
  - Note cycle in graph
    - \(P_1\) might request \(R_2\) before relinquishing \(R_1\)
    - Would cause deadlock

- **Detecting deadlock**
  - Static approaches (hard)
  - Program grinds to a halt
  - Threads package can keep track of locks held:
Fixing & debugging deadlocks

- Reboot system (windows approach)
- Examine hung process with debugger
- Threads package can deduce partial order
  - For each lock acquired, order with other locks held
  - If cycle occurs, abort with error
  - Detects potential deadlocks even if they do not occur
- Or use transactions...
  - Another paradigm for handling concurrency
  - Often provided by databases, but some OSES use them
  - Vino OS used transactions to abort after failures [Seltzer]

Transactions

- A transaction $T$ is a collection of actions with
  - Atomicity – all or none of actions happen
  - Consistency – $T$ leaves data in valid state
  - Isolation – $T$'s actions all appear to happen before or after every other transaction $T'$
  - Durability – $T$'s effects will survive reboots
  - Often hear mnemonic ACID to refer to above
- Transactions typically executed concurrently
  - But isolation means must appear not to
  - Must roll-back transactions that use others’ state
  - Means you have to record all changes to undo them
- When deadlock detected just abort a transaction
  - Breaks the dependency cycle

Transactional memory

- Some modern processors support transactional memory
- Transactional Synchronization Extensions (TSX) [intel1][15]
  - xbegin abort_handler – begins a transaction
  - xend – commit a transaction
  - xabort $code – abort transaction with 8-bit code
  - Note: nested transactions okay (also $xtest$ tests if in transaction)
- During transaction, processor tracks accessed memory
  - Keeps read-set and write-set of cache lines
  - Nothing gets written back to memory during transaction
  - On xend or earlier, transaction aborts if any conflicts
  - Otherwise, all dirty cache lines are written back atomically

Using transactional memory

- Use to get “free” fine-grained locking on a hash table
  - E.g., concurrent inserts that don’t touch same buckets are okay
  - Hardware will detect there was no conflict
- Use to poll for one of many asynchronous events
  - Start transaction
  - Fill cache with values to which you want to see changes
  - Loop until a write causes your transaction to abort
- Note: Transactions are never guaranteed to commit
  - Might overflow cache, get false sharing, see weird processor issue
  - Means abort path must always be able to perform transaction (e.g., you do need a lock on your hash table)

Hardware lock elision (HLE)

- Idea: have spinlocks that rarely need to spin
  - Begin a transaction when you acquire lock
  - Other CPUs won’t see lock acquired, can also enter critical section
  - Okay not to have mutual exclusion when no memory conflicts!
  - On conflict, abort and restart without transaction, thereby visibly acquiring lock (and aborting other concurrent transactions)
- Intel support:
  - Use $xacquire$ prefix before $xchgl$ (used for test and set)
  - Use $xrelease$ prefix before $movl$ that releases lock
  - Prefixes chosen to be noops on older CPUs (binary compatibility)
- Hash table example:
  - Use $xacquire$ xchg1 in table-wide test-and-set spinlock
  - Works correctly on older CPUs (with coarse-grained lock)
  - Allows safe concurrent accesses on newer CPUs!